A Proof Calculus for Attack Trees in Isabelle

Florian Kammüller

Middlesex University London and
Technische Universität Berlin
f.kammueller@mdx.ac.uk

Abstract. Attack trees are an important modeling formalism to identify and quantify attacks on security and privacy. They are very useful as a tool to understand step by step the ways through a system graph that lead to the violation of security policies. In this paper, we present how attacks can be refined based on the violation of a policy. To that end we provide a formal definition of attack trees in Isabelle's Higher Order Logic: a proof calculus that defines how to refine sequences of attack steps into a valid attack. We use a notion of Kripke semantics as formal foundation that then allows to express attack goals using branching time temporal logic CTL. We illustrate the use of the mechanized Isabelle framework on the example of a privacy attack to an IoT healthcare system.

1 Introduction

Identifying attacks and quantifying the attacker is a major challenge in security engineering. Attack trees are a simple classical approach but they still thrive in practical applications. One of the reasons is their simplicity and transparency to the user; the other is that their notion of attack analysis is a natural mechanism of a gradual approach to understanding security risks. In this paper, we provide a formal basis for attack trees in the interactive theorem prover Isabelle: a proof calculus for attack trees using a notion of refinement and attack validity. An existing emulation of modelchecking [6] provides a Kripke semantics for the proof calculus for attack trees. We introduce the proof calculus and the underlying mechanisation of the Kripke semantics. Finally, we illustrate the application of the presented Isabelle formalisation of attack trees on a case study from the health care sector which is the target of the CHIST-ERA project SUCCESS [3].

The main novelty of this paper is a mechanized theory for attack trees using Kripke structures to provide a state based foundation for the attack sequences as well as enabling the combination with the branching time logic CTL to facilitate detection and analysis of attacks.

The paper first introduces attack trees, Kripke structures, and attack tree refinement (Section 2) before presenting the proof calculus (Section 3). Section 4 then summarises the Isabelle Insider framework that can be used as an application of the attack tree formalisation. A health care system Insider attack is introduced and used as an illustrative example for the application of Isabelle attack trees and Kripke structures.

2 Attack Trees and Kripke Structures

2.1 Attack Trees

Attack Trees [16] are a graphical tree-based design language for the stepwise investigation and quantification of attacks. We believe that attack trees are a succinct way of representing attacks and thus not only useful as an immediate tool to quantify the attacker as part of a security analysis but also a good way of making security and privacy risks transparent to users. In attack trees [16,13], the root represents a goal, and the children represent sub-attacks. Sub-attacks can be alternatives for reaching the goal (disjunctive node) or they must all be completed to reach the goal (conjunctive node). Figure 1 illustrates the clarity of this graphical formalism by giving an example of an attack tree for opening a safe [16]. Leaf nodes represent the basic actions in an attack. Nodes of attack

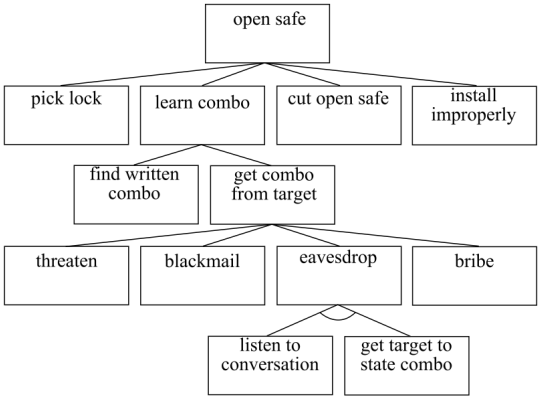

Fig. 1. Attack tree example illustrating mainly disjunctive nodes for alternative attacks refining the root node "open safe" and one conjunctive node for "eavesdrop".

trees can be adorned with attributes, for example costs of attacks or probabilities which allows quantification of attacks (not used in the example). Sub-trees can be combined disjunctively (or-nodes) or conjunctively (and-nodes).

As much as this clarity is encouraging to employ the formalism in the early stages of a security engineering process, it is also abstract and may lead to ambiguities. Therefore, it is desirable to lay foundations for attack trees that help us to use them not only to grasp intuitive attacks but to provide a foundation that helps to disambiguate and verify the intuition.

There are excellent foundations available based on graph theory [13]. They provide a very good understanding of the formalism, various extensions (like attack-defense trees [12] and differentiations of the operators (like sequential conjunction (SAND) versus parallel conjunction [5]) and are amply documented

in the literature. These theories for attack trees provide a thorough foundation for the formalism and its semantics. The main problem that adds complexity to the semantical models is the abstractness of the descriptions in the nodes. This leads to a variety of approaches to the semantics, e.g. propositional semantics, multiset semantics, and equational semantics for ADtrees [12]. The theoretical foundations allow comparison of different semantics, and provide a theoretical framework to develop evaluation algorithms for the quantification of attacks.

Surprisingly, the use of an automated proof assistant, like Isabelle, has not been considered despite its potential of providing a theory and analysis of attacks simultaneously. The essential attack tree mechanism of disjunction and conjunction in tree refinement is relatively simple. The complexity in the theories is caused by the attempt to incorporate semantics to the attack nodes and relate the trees to actual scenarios. This is why we consider the formalisation of a foundation of attack trees in the interactive prover Isabelle since it supports logical modeling and definitions of datatypes very akin to algebraic specification but directly supported by semi-automated analysis and proof tools.

2.2 Attack Tree Datatype in Isabelle

The attack trees formalisation including Kripke structures is formalised in Isabelle's Higher Order Logic. All sources are available online [7]. This Isabelle formalisation constitutes a tool for proving security properties using the assistance of the semi-automated theorem prover [11]. Isabelle is an interactive proof assistant based on Higher Order Logic (HOL). Applications can be specified as so-called object-logics in HOL providing reasoning capabilities for examples but also for the analysis of the meta-theory. An object-logic contains new types, constants and definitions. These items reside in a theory file, *e.g.*, the file `AT.thy` contains the object-logic for attack trees. This Isabelle Insider framework is a *conservative extension* of HOL. This means that an object logic does not introduce new axioms and hence guarantees consistency.

Attack trees have already been integrated as an extension to the Isabelle Insider framework [9,15] but with a limited scope to conjunctive nodes only and no added semantics to construct a proof calculus. In the current paper, we not only generalise the attack trees for arbitrary state systems but also properly extend to disjunctive nodes.

The principal idea is that base attacks are defined as a datatype and attack sequences as lists over them. Base attacks consist of actor's moves to locations, performing of actions and stealing of credentials stored at locations as expressed in the following datatype definition.

```
datatype baseattack = Goto "location"
                    | Perform "action"
                    | Credential "location"
```

The following datatype definition `attree` defines attack trees. The simplest case of an attack tree is a base attack. Attacks can also be combined as the conjunction or disjunction of other attacks. The operator $\oplus_\vee$ creates or-trees and $\oplus_\wedge$ creates

and-trees. And-attack trees $l \oplus^s_\wedge$ and or-attack trees $l \oplus^s_\vee$ combine lists of attack trees l either conjunctively or disjunctively on the attack goal s. The attack goal s is of arbitrary type α. It can be instantiated simply to the type `string` to represent the attack goal "informally" by an attack name. However, we can here also instantiate to a predicate type thereby enabling a constructive predicative description of the attack state using logic.

```
datatype attree = BaseAttack "baseattack" ("𝒩 (_)")
                | AndAttack "attree list "α"   ("_ ⊕∧(-)")
                | OrAttack  "attree list" "α"  ("_ ⊕∨(-)")
```

The functions `get_attseq` and `get_attack` are corresponding projections on attack trees returning the entire attack sequence or the final attack (the root), respectively. They are needed for defining the rule for attack refinement in Section 2.4.

2.3 Kripke Structures

Due to the expressiveness of Higher Order Logic (HOL), Isabelle allows us to formalise within HOL the notion of Kripke structures and temporal logic by directly encoding the fixpoint definitions for each of the CTL operators [6]. To realize this, a change of the considered system's state needs to be incorporated into Isabelle. A relation on system states is defined as an inductive predicate called `state_transition`. It introduces the syntactic infix notation `I` $\rightarrow_i$ `I'` to denote that system state `I` and `I'` are in this relation.

```
inductive state_transition ::  [state, state] ⇒ bool  ("_  →ᵢ _")
```

The definition of this inductive relation is given by a set of specific rules which are, however, not yet necessary to understand the notion of a Kripke structure and attack trees. They can be left out for the moment and will be introduced in Section 4.1, when we present the application of a healthcare Insider attack.

The set of states of a Kripke structure can be defined as the set of states reachable by the state transition from some initial state, for example, `Istate`.

```
Example_states ≡ { I. Istate →ᵢ^*  I }
```

The relation $\rightarrow_i$`^*` is the reflexive transitive closure – an operator supplied by the Isabelle theory library – applied to the relation $\rightarrow_i$.

The `Kripke` constructor combines a set of states, like the above example, and an initial state into a Kripke structure that is the graph formed by the closure over the state transition relation $\rightarrow_i$ starting in the initial state.

```
Example_Kripke ≡ Kripke Example_states {Istate}
```

When we now try to verify that some global security policy, say `global_policy`, holds for all paths globally in the example system, this can be expressed as follows in our Isabelle embedding of Kripke structures and branching time logic CTL [6].

```
Example_Kripke ⊢ AG global_policy
```

The relation $\rightarrow_i$ provides a transition between states of a system. State transitions transform a state into another state by actions that change this state. In the human centric systems that we focus on, these actions are executed by actors. By contrast for attack trees, we have not yet explicitly introduced an effect on the system's state but we equally investigate and refine attacks as sequences of actions eventually mapping those actions onto sequences of base attacks. In the current approach, we use the Kripke models as the *semantics* for the attack tree analysis. More precisely, the sequences of attack steps that are eventually found by the process of refining an attack, need to be checked against sequences of state transitions possible in the Kripke structure that consists of the graph of system state changes.

Technically, we need a slight transformation between sequences of steps of the system's state changing relation $\rightarrow_i$ and sequences of actions of actors leading to states where policies are violated. We simply annotate the state transitions by actions. Then, sequences of actions naturally correspond to the paths that determine the way through the Kripke structure and can be one-to-one translated into attack vectors.

Formally, we simply define a relation very similar to $\rightarrow_i$ but with an additional parameter added as a superscript after the arrow.

```
inductive state_step ::  [state, action, state] ⇒ bool  ("_  →(-) _")
```

We define an iterator relation `state_step_list` over the `state_step` that enables collecting the action sequences over state transition paths.

```
inductive state_step_list :: [state, action list, state] ⇒ bool
             ("_  →(-) _")
where
 state_step_list_empty: I →[] I |
 state_step_list_step : ⟦ I →[a] I'; I' →l I'' ⟧
                        ⟹ I →a#l I''
```

With this extended relation on states we can now trace the action sequences. Finally, a simple translation of attack sequences from the attack tree model to action sequences can simply be formalised by first defining a translation of base attacks to actions.

```
primrec transform :: baseattack ⇒ action
where
 transform_move:    transform (Goto l') = move |
 transform_get:     transform (Credential l') = get |
 transform_perform: transform (Perform a) = a
```

From this we define a function `transf` for transforming sequences of attacks.

```
primrec transf :: baseattack list ⇒ action list
where
 transf_empty : transf [] = [] |
 transf_step:   transf (ba#l) = (transform ba)#(transf l)
```

2.4 Attack Refinement

The main construction concept for attack trees is *refinement* defined by an inductive predicate `refines_to` syntactically represented as the infix operator $\sqsubseteq$. Intuitively, refinement corresponds to developing an attack tree from the root to the leaves (see Figure 2). Refinement is an order relation on sub-trees of an attack tree formalising this intuition. There are rules `trans` and `refl` making the refinement a preorder; the rule `refineI` shows how attack vectors can be integrated into the refinement process by extending an abstract attack into a conjunctive sequence of more concrete attacks. The term `sublist_rep l a (get_attseq A)` replaces an attack `a` by the attack sequence `l` in the attack sequence of attack tree `A` given by its leaves. The definition of this function is a straightforward recursive list function and omitted here for brevity [7]. The rule `refineO` defines how an attack `A` can be refined into a disjunction of attacks `as` if each of these attacks refines `A`. The complete definition of the inductive definition of attack tree refinement is given in Table 1.

inductive `refines_to :: [attree, state, attree] ⇒ bool  ("_ ⊑(-) _")`
where
`refineI:` $[\![$ `I` $\rightarrow_i^*$ `I'; I'` $\rightarrow^{l'}$ `I''; transf l = l';`
`sublist_rep l a (get_attseq A) = (get_attseq A');`
`get_attack A = get_attack A' ]]` $\Longrightarrow$ `A` $\sqsubseteq_I$ `A' |`
`refineO:` $\forall$ `A' ∈ set(as). A` $\sqsubseteq_I$ `A' ∧ get_attack A = s` $\Longrightarrow$ `A` $\sqsubseteq_I$ `as` $\oplus^s_\vee$ `|`
`trans:` $[\![$ `A` $\sqsubseteq_I$ `A'; A'` $\sqsubseteq_I$ `A'' ]]` $\Longrightarrow$ `A` $\sqsubseteq_I$ `A'' |`
`refl :` `A` $\sqsubseteq_I$ `A`

Table 1. Attack tree refinement: inductive definition containing defining rules.

An application can be seen in Section 4.3 where we apply the attack tree analysis to the health care case study.

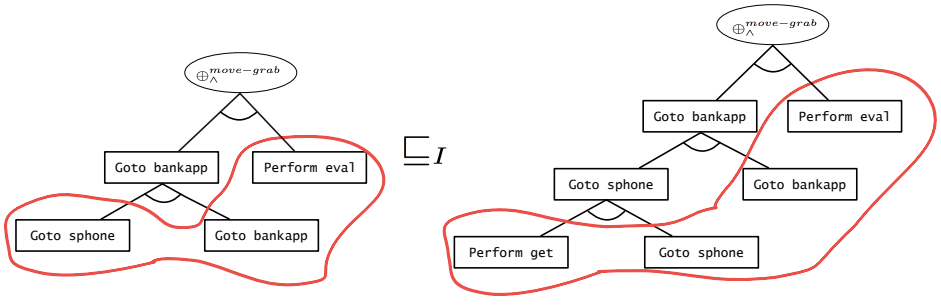

Fig. 2. Attack refinement for healthcare case study (see also Section 4.3).

The refinement of attacks allows the expansion of top level abstract attacks into longer sequences or disjunctions. Ultimately, we need to have a notion of when a sufficiently refined sequence of attacks is valid. This notion is provided by the *proof calculus* for attack trees which allows the deduction of validity of attacks expressed formally as $I, h \vdash a$ saying that in the state I the actor h can perform attack a. The proof calculus integrates attack tree refinement and is presented in the following Section 3.

3 Proof Calculus

The proof calculus for attack trees provides a notion of validity of an attack tree with respect to a given system and an attacker. The definition of the proof calculus for attack trees is given in Tables 2 and 3.

For individual attack steps, it presupposes a definition of the behaviour of an attacker in a system given by the `enables` predicate to set off the derivation of valid base attacks (rules `att_act, att_goto, att_cred`). This `enables` predicate is treated here as an abstract predicate over the state describing whether an actor is entitled by the policy to execute a specific action. In the application example in the following section, we will see an example for a concrete definition for this enables predicate in the Isabelle Insider framework.

The rule `att_ref` states that an abstract attack that can be refined into a valid concrete attack is itself valid. The rule `att_comp_and` defines how an attack `as` $\oplus^{s'}_{\wedge}$ can be conjoined with a valid conjunctive attack `as'` $\oplus^{s}_{\wedge}$ into a larger conjunctive attack `as @ as'` $\oplus^{s}_{\wedge}$. The operator `@` is the Isabelle list operator for appending two lists. In this rule, the system state `I` before the first attack needs to allow a state transition `I` $\rightarrow^*$ `I'` to the state `I'` before the second attack. Since Isabelle is a Higher Order Logic theorem prover, the variables `I, I'` are higher order variables. This permits a flexible instantiation within a derivation and a gradual development of concrete states that exhibit corresponding pre-conditions and post-conditions of attacks. Since we use the reflexive transitive closure $\rightarrow^*$ (available in Isabelle as a constructor of relations) the rule also allows the pre-states and post-states `I, I'` to be identical. Thus, we can in one rule express sequential and concurrent conjunctive attacks. We do not need a separate rule for SAND as in other foundations for attack trees, e.g. [5]. The rule for disjunctive composition uses universal quantification to express that a list of disjunctive attacks needs to have the same pre-state and post-state (these states `I, I'` are fixed by the same quantifier) in order to be unified in an "or" attack tree. The rule `att_comp_and` defines how two and-sequences of attacks can be added to one larger attack.

As a consequence of introducing also or-attacks for attack trees, we naturally create the need to define how or-attacks and and-attacks relate to each other. We therefore extend the inductive definition with the distribution rules presented in Table 3.

An advantage of using an interactive theorem prover like Isabelle is that the rules of the inductive definition can be used to derive within the theorem

```
inductive is_and_attack_tree :: [state, actor, attree] ⇒ bool
        ("_, _ ⊢ _")
 where
  att_act: enables I l h a ⟹ I , h ⊢ 𝒩(Perform(a)) |
  att_goto: enables I l h (move) ⟹ I, h ⊢ 𝒩(Goto l) |
  att_cred: enables I l h (get) ⟹ I, h ⊢  𝒩(Credential l) |
  att_ref: ⟦ A ⊑_I A'; I, h ⊢ A' ⟧ ⟹ I, h ⊢ A  |
  att_and_one: I, h ⊢ a  ⟹ I, h ⊢ [a] ⊕^s_∧  |
  att_comp_and: ⟦ I, h ⊢  as ⊕^s'_∧; I →* I'; I', h ⊢  as' ⊕^s_∧ ⟧
               ⟹ I, h ⊢ as @ as' ⊕^s_∧ |
  att_comp_or:  ⟦ ∀ a ∈ (set(as)). I, h ⊢ a ∧ get_attack a = s ⟧
               ⟹ I, h ⊢ as ⊕^s_∨
  ...
```

Table 2. Proof calculus for attack trees: main part

```
...
att_and_distr_left: I, h ⊢ ( [a,(as ⊕^s_∨)] ⊕^s_∧)
              ⟹  I, h ⊢ ((map (λ x. [a, x]⊕^s_∧) as) ⊕^s_∨) |
att_and_distr_right: I, h ⊢ ( [(as ⊕^s_∨),a] ⊕^s_∧)
              ⟹  I, h ⊢ ((map (λ x. [x, a] ⊕^s_∧) as) ⊕^s_∨) |
att_or_distr_left: I, h ⊢ ((map (λ x. [a, x]⊕^s_∧) as) ⊕^s_∨)
              ⟹ I, h ⊢ ( [a,(as ⊕^s_∨)] ⊕^s_∧) |
att_or_assoc_right: I, h ⊢ ((map (λ x. [x, a] ⊕^s_∧) as) ⊕^s_∨)
              ⟹ I, h ⊢ ( [(as ⊕^s_∨),a] ⊕^s_∧)
```

Table 3. Proof calculus for attack trees: distributivity rules

prover. This avoids introducing inconsistencies but in general also enables the development of meta-theory, i.e., theoretical consequences of the definitions of the concepts, here attack trees. For example, standard rules, like associativity rules, for attack trees can be derived. But also other rules, like for example a "one-step" composition rule for and-attacks adding just a single attack `a` at the front of an attack sequence `as` using the cons-operation `#` on lists.

lemma `att_comp_and_cons:` $[\![$ `I, h ⊢ a ; I', h ⊢ as` $\oplus^s_\wedge$`; I` $\rightarrow^*$ `I'` $]\!]$
$\Longrightarrow$ `(I, h ⊢ (a # as)` $\oplus^s_\wedge$`)`

In this paper, we base the definitions of system, actors, their behaviour, and the corresponding state transitions on the Isabelle Insider framework. The presented proof calculus for attack trees is easily applicable to other models of applications by exchanging the behaviour predicate and using the corresponding state transition relation. The calculus only considers attacks by single actors. An extension to sets of actors can be defined in a straightforward manner based on this calculus.

4 Application: Insider Attack in IoT Healthcare

In this section, we finally illustrate how the proof calculus for attack trees is applied to an example. We instantiate the formalism to the Isabelle Insider framework that supports the representation of infrastructures as graphs with actors and policies attached to nodes. These infrastructures are the *states* of the Kripke structure for the attack trees. This section gives a brief summary of the main relevant parts of the Isabelle Insider framework: actions, actors, infrastructures, behaviour and state transition relation. We next give a summary of our health care case study before illustrating how the attack tree analysis is performed on it using the attack tree mechanism.

4.1 Isabelle Insider framework

The Isabelle Insider framework [11] is based on a logical process of sociological explanation [4] inspired by Weber's *Grundmodell*, to explain Insider threats by moving between societal level (macro) and individual actor level (micro).

The interpretation into a logic of explanation is formalized in the Isabelle Insider framework [11]. The micro-level and macro-level of the sociological explanation give rise to a two-layered model in Isabelle, reflecting first the psychological disposition and motivation of actors and second the graph of the infrastructure where nodes are locations with actors associated to them. Security policies can be defined over the agents, their properties, and the infrastructure graph; properties can be proved mechanically with Isabelle.

In the Isabelle/HOL theory for Insiders, one expresses policies over actions `get`, `move`, `eval`, and `put`. The framework abstracts from concrete data – actions have no parameters:

datatype `action = get | move | eval | put`

The human component is the *Actor* which is represented by an abstract type `actor` and a function `Actor` that creates elements of that type from identities (of type `string`):

```
typedecl actor
type_synonym identity = string
consts Actor :: string ⇒ actor
```

Policies describe prerequisites for actions to be granted to actors given by pairs of predicates (conditions) and sets of (enabled) actions:

```
type_synonym policy = ((actor ⇒ bool) × action set)
```

Policies are integrated with a graph into the infrastructure providing an organisational model where policies reside at locations and actors are adorned with additional predicates to specify their 'credentials', and a predicate over locations to encode attributes of infrastructure components:

```
datatype infrastructure = Infrastructure
                  "igraph" "location ⇒ policy set"
                  "actor ⇒ bool" "location ⇒ bool"
```

These local policies serve to provide a specification of the 'normal' behaviour of actors but are also the starting point for possible attacks on the organisation's infrastructure. The `enables` predicate specifies that an actor `a` can perform an action `a'`∈ `e` at location `l` in the infrastructure `I` if `a`'s credentials (stored in the tuple space `tspace I a`) imply the location policy's (stored in `delta I l`) condition `p` for `a`:

```
enables I l a a' ≡ ∃ (p,e) ∈ delta I l. a' ∈ e
               ∧ (tspace I a ∧ lspace I l ⟶ p(a))
```

This definition of the behaviour for the Insider framework allows to define the rules for the state transition relation of the Kripke structure (see Section 2.3) for each of the actions. Here is the rule for move.

```
move: ⟦ G = graphI I; a @_G l; l ∈ nodes G;
        l' ∈ nodes G; a ∈ actors_graph(graphI I);
        enables I l (Actor a) move;
        I' = Infrastructure (move_graph_a a l l'
              (graphI I))(delta I)(tspace I)(lspace I)
      ⟧ ⟹ I →_i I'
```

4.2 Health Care Case Study

The case study we use as a running example in this paper is a simplified scenario from the context of the SUCCESS project for Security and Privacy of the IoT [3]. A central topic of this project for the pilot case study is to support security and privacy when using cost effective methods based on the IoT for monitoring patients for the diagnosis of Alzheimer's disease. As a starting point for the

design, analysis, and construction, we currently develop a case study of a small device for the analysis of blood samples that can be directly connected to a mobile phone. The analysis of this device can then be communicated by a dedicated app on the smart phone that sends the data to a server in the hospital.

In this simplified scenario, there are the patient and the carer within a room together with the smart phone.

We focus on the carer having access to the phone in order to support the patient in handling the special diagnosis device, the smart phone, and the app.

The insider threat scenario has a second banking app on the smart phone that needs the additional authentication of a "secret key": a small electronic device providing authentication codes for one time use as they are used by many banks for private online banking.

Assuming that the carer finds this device in the room of the patient, he can steal this necessary credential and use it to get onto the banking app. Thereby he can get money from the patient's account without consent.

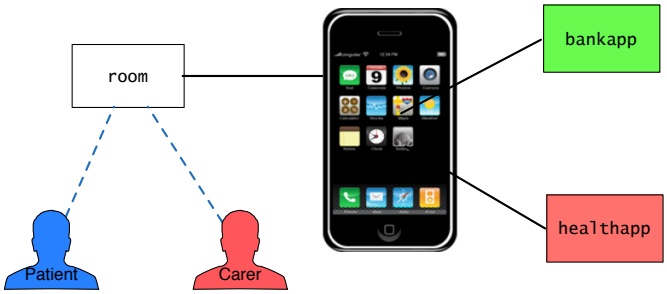

Fig. 3. Health care scenario: carer and patient in the room may use smartphone apps.

4.3 Health Care Case Study in Isabelle Insider Framework

We only model two identities, `Patient` and `Carer` representing a patient and his carer. We define the health care scenario in the locale `scenarioHealthcare`. The syntax `fixes` and `defines` are keywords of locales that we drop together with the types for clarity of the exposition from now on. The double quotes `''s''` represent strings in Isabelle/HOL. The global policy is 'no one except the patient can use the bank app':

```
fixes global_policy ::  [infrastructure, identity] ⇒ bool
defines  global_policy I a ≡  a ≠ ''Patient'' ⟶
                ¬(enables I bankapp (Actor a) eval)
```

The graph representing the infrastructure of the health care case study has the following locations: (0) smart phone, (1) room, (2) bank app, and (3) health app:

In order to define the infrastructure, we first define the graph representing the scenario's locations and the positions of its actors. The actors patient and carer are both initially in room. The graph is given as a set of nodes of locations and the actors residing at certain locations are specified by a function associating lists of nodes with the locations.

```
ex_graph ≡
 Lgraph {(room, sphone), (sphone, healthapp),
         (sphone, bankapp)}
         (λ x. if x = room then
               [''Patient'', ''Carer''] else [])
```

In the following definition of local policies for each node in the office scenario, we additionally include the parameter `G` for the graph. The predicate $@_G$ checks whether an actor is at a given location in the graph G.

```
local_policies G  ≡
 (λ x. if x = room then {(λ y. True,{get, put, move})}
   else (if x = sphone then
    {((λ y. has (y, ''PIN'')), {put,get,eval,move}), (λ y. True, {})}
        else (if x = healthapp then
         {((λ y. (∃ n. (n @_G sphone) ∧ Actor n = y)),
           {get,put,eval,move})}
             else (if x = bankapp then
              {((λ y. (∃ n. (n @_G sphone) ∨ (n @_G bankapp)
                ∧ Actor n = y ∧ has (y, ''skey''))),
                {get,put,eval,move})}
                else {}))))
```

In this policy, any actor can move to the room and when in possession of the PIN can move onto the sphone and do all actions there. The following restrictions are placed on the two other locations.

`healthapp`: to `move` onto the `healthapp` and perform any action at this location, an actor must be at the position `sphone` already;
`bankapp`: to `move` onto the `bankapp` and perform any action at this location, an actor must be at the position `sphone` already and in possession of the `skey`.

The possession of credentials like PINs or the skey is assigned in the infrastructure as well as the roles that actors can have. We define this assignment as predicate over actors being true for actors that have these credentials. For the health care scenario, the credentials express that the actors `Patient` and `Carer` possess the PIN for the sphone but Patient also has the skey.

```
ex_creds ≡
  (λ x. if x = Actor ''Patient'' then
                has (x,''PIN'') ∧ has (x, ''skey'')
          else (if x = Actor ''Carer'' then
                has (x, ''PIN'') else True))
```

The graph and credentials are put into the infrastructure `hc_scenario`.

```
hc_scenario ≡ Infrastructure
                ex_graph (local_policies ex_graph)
                ex_creds ex_locs
```

4.4 Attack Tree Analysis

System states in the application to the Insider framework are given by infrastructures. The initial state corresponds to the above `hc_scenario`; following states are introduced by applying the state transition function. We introduce the following definitions to denote changes to the infrastructure. A first step towards critical states is that the carer gets onto the smart phone. We first define the changed infrastructure graph.

```
ex_graph' ≡  Lgraph
             {(room, sphone), (sphone, healthapp),
               (sphone, bankapp)}
             (λ x. if x = room
                   then [''Patient''] else
                   (λ x. if x = sphone
                         then [''Carer''] else []))
```

The dangerous state has a graph in which the actor `Carer` is on the bankapp.

```
ex_graph'' ≡  Lgraph
              {(room, sphone), (sphone, healthapp),
                (sphone, bankapp)}
              (λ x. if x = room
                    then [''Patient''] else
                    (λ x. if x = bankapp
                          then [''Carer''] else []))
```

The critical state of the credentials is where the carer has the skey as well.

```
ex_creds' ≡
  (λ x. if x = Actor ''Patient'' then
              has (x,''PIN'') ∧ has (x, ''skey'')
             else (if x = Actor ''Carer'' then
                has (x, ''PIN'') ∧ has (x, ''skey'')
                   else True))
```

We use these changed state components to define a series of infrastructure states.

```
hc_scenario'  ≡ Infrastructure
                ex_graph (local_policies ex_graph)
                ex_creds' ex_locs
hc_scenario'' ≡ Infrastructure
                ex_graph'(local_policies ex_graph')
                ex_creds' ex_locs
hc_scenario'''≡ Infrastructure
                ex_graph''(local_policies ex_graph'')
                ex_creds' ex_locs
```

We next look at the abstract attack that we want to analyse before we see how Kripke structures and temporal logic support the analysis.

The abstract attack is stated as `[Goto bankapp, Perform eval]` $\oplus_{\wedge}^{move-grab}$. The following refinement encodes a logical explanation of how this attack can happen by the carer taking the skey, getting on the phone, on the bankapp and then evaluating.

```
[Goto bankapp,  Perform eval] ⊕∧^move−grab
⊑hc_scenario
[Perform get, Goto sphone, Goto bankapp, Perform eval] ⊕∧^move−grab
```

This refinement is proved by applying the rule `refineI` (see Section 2.4). In fact, this attack could be *found* by applying `refineI` and using interactive proof with Isabelle to instantiate the higher order parameter `?l` in the following resulting subgoal.

```
hc_scenario →^transf(?l) hc_scenario'''
```

This proof results in instantiating the variable `?l` to the required attack sequence `[Perform get, Goto sphone, Goto bankapp, Perform eval]`.

So far, we have used the combination of a slightly adapted notion of the state transition of the Kripke structures to build a model for attack refinement of attack trees. We can further use the correspondence between Kripke structures and attack trees to find attacks. We first define the Kripke structure for the health case scenario representing the state graph of all infrastructure states reachable from the initial state.

```
hc_states ≡ { I. hc_scenario →i* I }
hc_Kripke ≡ Kripke hc_states {hc_scenario}
```

Since it is embedded into Isabelle [6], we may use branching time logic CTL to express that the global policy (see Section 4.3) holds for all paths globally.

```
hc_Kripke ⊢  AG {x. global_policy x ''Carer''}
```

Trying to prove this must fail. However, using instead the idea of invalidation [10] we can prove the negated global policy.

```
hc_Kripke ⊢  EF {x. ¬ global_policy x ''Carer''}
```

The interactive proof of this `EF` property means proving the theorem

```
hc_Kripke ⊢  EF {x. enables x bankapp
                              (Actor ''Carer'') eval}
```

This results in establishing a trace `l` that goes from the initial state `hc_scenario` to a state `I` such that `enables I bankapp (Actor ''Carer'') eval`. This `I` is for example `hc_scenario'''` and the action path `get, move, move` is a side product of this proof. Together with the states on this path the `transf` function delivers the required attack path `[Perform get, Goto sphone, Goto bankapp, Perform eval]`.

5 Conclusion

Summarizing, we have provided a mechanized foundation for attack trees. The semantics of attack trees has been defined using an embedding of modelchecking in Isabelle leading to a proof calculus for attack trees. We illustrated the benefits on a health care case study of an Insider attack using the semantics on the Isabelle Insider framework infrastructures as our system state but this state model can be replaced by other suitable state models to apply Isabelle attack trees and Kripke structures.

There is a range of observations concerning the relation between attack trees and Kripke structures in Isabelle that we presented in this paper and whose conception, construction, and demonstration represents our contribution.

- Kripke structures can be used as the underlying semantics for state based systems interpreting the attacks, i.e., providing semantics for attack trees.
- Therefore, the state transition relation can be used to define refinement steps in the refinement part of a proof calculus for attack trees.
- Higher Order Logic variables for pre-states and post-states of an attack step can be dynamically derived in applications of our proof calculus.
- Temporal logic formulas in the branching time logic CTL can be used in our Isabelle framework extension supporting the detection of attacks.
- The attack tree proof calculus serves as a logical basis to judge the validity of an attack in a given model.
- The attack tree proof calculus can be applied to case studies as demonstrated on an IoT health care application case study.

Clearly relevant to this work are the Isabelle Insider framework and its extensions [11,9,8,6] but also the related experiments with the invalidation approach for Insider threat analysis using classic implementation techniques like static analysis and implementation in Java [15] or probabilistic modeling and analysis [2].

We believe that the combination of Kripke structures and attack trees is novel in the way we tie these concepts up at the foundational level. Considering the simplicity of this pragmatically driven approach and the relative ease with which we arrived at convincing results, it seems a fruitful prospect to further explore this combination. Beyond the mere finding of attack vectors in proofs, the expressivity of Higher Order Logic will allow developing meta-theory that in turn can be used for the transfer between state based reasoning and attack tree analysis.

The presented foundation of attack trees in Isabelle is consistent with the existing foundations [13,12,5] but instead of providing an on paper mathematical foundation it provides a direct formalisation in Higher Order Logic in the proof assistant. This enables the application of the resulting framework to case studies and does not necessitate a separate implementation of the mathematical foundation in a dedicated tool. Clearly, the application to case studies requires user interaction. However, the formalisation in Isabelle supports not only the application of the formalised theory but furthermore the consistent development of

meta-theorems thus guaranteeing consistency at all levels. In addition, dedicated proof automation by additional proof of supporting lemmas is straightforward and even code generation is possible for executable parts of the formalisation.

In comparison to the existing foundations [13,12,5], the presented attack tree framework only covers a portion of available extensions for attack trees. For example, it does not support attack-defense trees, i.e., the integration of defenses within the attack tree. This is a straightforward future development. Other work on attack trees includes the extension of the formalism by probabilities and time [1]. To support this quantitative analysis, automated verification techniques using modelchecking with the UPPAAL system and timed automata are applied as well [14]. This direct application of modelchecking provides automated analysis of attack trees but unlike our proof theory for attack trees it does not allow any proofs about attack trees. Thereby, the consistency and partially also the adequacy of the model is not guaranteed. However, we believe that a complementary use of these works with our more expressive formalisation is fruitful for developing secure systems from early requirements.

References

1. F. Arnold, H. Hermanns, R. Pulungan, and M. Stoelinga. Time-dependent analysis of attacks. In *Principles of Security and Trust, POST'14*, LNCS, pages 285–305. Springer, 2014.
2. T. Chen, F. Kammüller, I. Nemli, and C. W. Probst. A probabilistic analysis framework for malicious insider threats. In T. Tryfonas and I. G. Askoxylakis, editors, *Human Aspects of Information Security, Privacy, and Trust - Third International Conference, HAS 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015. Proceedings*, volume 9190 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2015.
3. CHIST-ERA. Success: Secure accessibility for the internet of things, 2016. http://www.chistera.eu/projects/success.
4. C. G. Hempel and P. Oppenheim. Studies in the logic of explanation. *Philosophy of Science*, 15:135–175, April 1948.
5. R. Jhawar, B. Kordy, S. Mauw, S. Radomirovic, and R. Trujillo-Rasua. Attack trees with sequential conjunction. In *30th IFIP TC 11 International Conference on ICT Systems Security and Privacy Protection (IFIP SEC'15)*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 339–353. Springer, 2015.
6. F. Kammüller. Isabelle modelchecking for insider threats. In *Data Privacy Management, DPM'16, 11th Int. Workshop, co-located with ESORICS'16*, volume 9963 of *LNCS*. Springer, 2016.
7. F. Kammüller. Isabelle insider framework with Kripke structures, CTL, attack trees and refinement, 2017. Available from `https://www.dropbox.com/sh/rx8d09pf31cv8bd/AAALKtaP8HMX642fi04Og4NLa?dl=0`.
8. F. Kammüller, M. Kerber, and C. Probst. Towards formal analysis of insider threats for auctions. In *8th ACM CCS International Workshop on Managing Insider Security Threats, MIST'16*. ACM, 2016.
9. F. Kammüller, J. R. C. Nurse, and C. W. Probst. Attack tree analysis for insider threats on the IoT using Isabelle. In *Human Aspects of Information Security,*

Privacy, and Trust - Fourth International Conference, HAS 2015, Held as Part of HCI International 2016, Toronto, Lecture Notes in Computer Science. Springer, 2016. Invited paper.
10. F. Kammüller and C. W. Probst. Invalidating policies using structural information. In *IEEE Security and Privacy Workshops (SPW)*. IEEE, 2013.
11. F. Kammüller and C. W. Probst. Modeling and verification of insider threats using logical analysis. *IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence*, 11:534–545, June 2017 2017.
12. B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Attack-defense trees. *Journal of Logic and Computation*, 24(1):55–87, 2014.
13. B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13–14:1–38, 2014.
14. R. Kumar, E. Ruijters, and M. Stoelinga. Quantitative attack tree analysis via priced timed automata. In *FORMATS 2015*, pages 156–171, 2015.
15. C. W. Probst, F. Kammüller, and R. R. Hansen. Formal modelling and analysis of socio-technical systems. In *Semantics, Logics, and Calculi (Nielsens' Festschrift)*, volume 9560 of *LNCS*. Springer, 2016.
16. B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2004.